53405

Charon toolkit for parallel, implicit structured-grid computations: Functional design

Rob F. Van der Wijngaart
MRJ Technology Solutions
NASA Ames Research Center
Moffett Field, CA 94035

1 Introduction

In a previous report the design concepts of *Charon* [5] were presented. Charon is a toolkit that aids engineers in developing scientific programs for *structured-grid* applications to be run on MIMD parallel computers. It constitutes an augmentation of the general-purpose MPI-based [4] message-passing layer, and provides the user with a hierarchy of tools for rapid prototyping and validation of parallel programs, and subsequent piecemeal performance tuning.

Here we describe the implementation of the domain decomposition tools used for creating data distributions across sets of processors. We also present the hierarchy of parallelization tools that allows smooth translation of legacy code (or a serial design) into a parallel program. Along with the actual tool descriptions, we will present the considerations that led to the particular design choices. Many of these are motivated by the requirement that Charon must be useful within the traditional computational environments of Fortran 77 and C. Only the Fortran 77 syntax will be presented in this report.

2 A multi-level, orthogonal design

In Charon we distinguish between data distribution support and parallel execution support. They are orthogonal elements of the design space, meaning that high-level data distribution functions can be applied in conjunction with low-level parallel execution support tools, and vice versa. Much of the trouble in the implementation of advanced algorithms on MIMD message-passing systems stems from the fact that data distribution and concurrent execution on distributed data structures cannot be separated. Those systems that do allow the separation between the two are usually very restricted in both the type of operations and the type of distributions allowed, as was detailed in [5]. Charon is able to provide virtually complete freedom in data distribution while still offering powerful support tools for the control of the program flow, because it treats parallelization as an incremental process, whose final product must be very efficient, but whose intermediate stages are allowed to be slow. The foremost decision, made by the user, is the choice of the data distribution (possibly dynamic). Whereas the lowest level of abstraction in Charon features close integration of data distribution and parallel execution environment, the highest level of abstraction contains support tools that allow the program to execute correctly on a distributed data set, with minimal structural and textual changes in the serial program text. This is accomplished as follows.

All arrays that need to be distributed register with a distribution utility that structures the local storage space for a segment (or segments) of the the array (see Section 3). All other variables are global, which means that they exist on all processors, and have the same value on all. Moreover, all processors execute the same program statements. Whenever an assignment to an element of a distributed array takes place, the owner-computes rule is invoked, which means that only one processor performs the actual assignment. Whenever an assignment requires the value of a distributed array element, a communication takes place automatically if the element is remote. The mechanism for such flexible assignments, which makes use of the Charon functions `assign`, `address`, and `value`, is described in Section 4. Complications may occur, for example when a user function is called that takes as input distributed array elements, and uses their addresses to modify nearby memory locations without providing explicit information regarding the position of those locations within the distributed array. In that case more versatile access mechanisms need to be applied that (may) fetch blocks of data, and that regulate execution of functions depending on which processor owns the data being modified. For this purpose the Charon functions `invoke` and `mvalue` are provided. Despite this flexibility, there are still certain program structures that cannot be expressed using the top-level functions in Charon. That is because those functions rely on *atomicity* of the owner-computes rule. Atomicity is automatically satisfied by a single assignment, but can be violated by user-defined or library functions operating on pointers. If a function modifies multiple distinct distributed arrays, or different parts of the same distributed array, there can be a conflict about ownership of the data to be modified. It is the responsibility of the user to resolve such conflicts, or to guarantee program correctness implicitly.

3 Distribution support tools

Charon supports the parallelization of programs using multi-dimensional arrays related to structured grids. The data distribution process consists of three fundamental steps:

1. Define a *grid* and create a partitioning using Cartesian sections. The result is called a *partition*.

2. Assign partitions to processors. The result is called a *decomposition*.

3. Create the multi-dimensional, distributed array and associate it with a decomposition. The result is called a distributed grid variable, or *distribution*.

Common decompositions, such as uni-partitions or diagonal multi-partitions (see [5, 6]), can be created with a few high-level decomposition functions. Customizations are performed using lower-level functions. In the following description of Charon functions, the integer variables `grid`, `partition`, `decomposition`, and `distribution` (in typewriter font) are handles to the corresponding data structures.

create_grid, set_grid_size, and set_grid_start are used to define a discretization grid of a certain dimensionality, to specify the size of the grid in a particular dimension, and to redefine the starting index of the grid in a specific dimension (default is specified through the Charon function set_default_offset; see below), respectively. The grid and all subsequent constructs based on it are restricted to the processors in the MPI communicator specified in create_grid.

Syntax of Fortran 77 grid creation functions.

```
subroutine create_grid(grid,communicator,no_dimensions)
integer communicator, grid, no_dimensions

subroutine set_grid_size(grid,dimension,size)
integer grid, dimension, size

subroutine set_grid_start(grid,dimension,start)
integer grid, dimension, start
```

Create_partition, set_no_cuts, set_cut, and set_even_cuts are used to define a partition, to specify the number of cuts in a certain dimension, to (re)define the value of a particular cut (a value of n means that a separator is placed between points $n-1$ and n), and to space cuts in a certain dimension as evenly as possible, respectively. If no exactly uniform division is possible, set_even_cuts will augment the size of the low numbered partitions by one unit until the leftover points have been exhausted.

Syntax of Fortran 77 partitioning functions:

```
subroutine create_partition(partition,grid)
integer partition, grid

subroutine set_no_cuts(partition,dimension,no_cuts)
integer partition, dimension, no_cuts

subroutine set_cut(partitio,dimension,cut,value)
integer partition, dimension, cut, value

subroutine set_even_cuts(partition,dimension)
integer partition, dimension
```

create_decomposition and set_owner are used to define a decomposition and to (re)set ownership of a particular partition, respectively. Ownership is signified by the rank of the processor within the communicator. Because grids can have different dimensionality, the number of indices needed to identify a partition can vary. It is the user's responsibility to supply the correct number.

Syntax of Fortran 77 decomposition functions:

```
subroutine create_decomposition(decomposition,partition)
integer decomposition, partition
```

```
subroutine set_owner(decomposition,owner_rank,index1,index2,...)
integer decomposition, owner_rank, index2, index2, ...
```

Note that Fortran 77 requires a fixed number of parameters for each function or subroutine. The definition of `set_owner` and several other Charon functions does not conform to that standard, which may lead to problems on some computer systems. These can be resolved by using a Fortran 90 compiler instead, which will automatically insert default values for the dummy indices that are left unspecified. Whereas the `set_owner` routine suffices to construct any type of decomposition, it is usually preferable to create commonly used decompositions using a few high-level routines. It is simpler, less error-prone, and also more efficient; Charon can use optimized interrogation and communication calls when the decomposition has a known, regular structure. The common decompositions currently supported by Charon are uni-partitions (each processor is assigned a single partition), diagonal multi-partitions (each processor is assigned several partitions in a regular pattern [6]), and High Performance Fortran-style [2] block-cyclic distributions. The predefined decompositions assume partitioning of the grid in all dimensions. However, particular dimensions can be excluded by invoking the command `exclude_partition_dimension`. In multi-partition decompositions at least two grid dimensions must be partitioned. By default, the uni-partition decomposition minimizes aspect ratios in the partitioned dimensions (for example, a grid of 80 × 20 points would be divided in 16 partitions of 10 × 10 points). Customization is obtained by specifying the number of processors in each particular grid dimension (`set_no_processors`). By default, the block-cyclic decomposition uses a cyclic distribution with a group size of one in all partitioned dimensions. The number of processors to be applied to each partitioned grid dimension is as close to equal as possible. Different numbers of processors and different group sizes can be specified using `set_no_processors` and `set_group_size`, respectively. In addition, a block decomposition can be specified for selected dimensions using `set_block_partition`. When a decomposition is constructed using high-level Charon functions, a partition is created implicitly. The function `partition` returns a handle to that partition.

Syntax of Fortran 77 high-level decomposition functions:

```
subroutine create_unipartition(decomposition,grid)
integer decomposition, grid

subroutine create_multipartition(decomposition,grid)
integer decomposition, grid

subroutine create_block_cyclic(decomposition,grid)
integer decomposition, grid

subroutine exclude_partition_dimension(decomposition,dimension)
integer decomposition, dimension

subroutine set_no_processors(decomposition,dimension,no_procs)
integer decomposition, dimension, no_procs)
```

```
subroutine set_group_size(decomposition,dimension,size)
integer decomposition, dimension, size)

subroutine set_block_partition(decomposition,dimension)
integer decomposition, dimension

integer function partition(decomposition)
integer decomposition
```

Note that the numbering of all array elements requires the definition of an array offset. For Fortran the default offset is 1. For C and C++ it is 0. Consequently, the first grid dimension has index 1 in Fortran, and index 0 in C. The default offset can be changed by calling `set_default_offset(offset)`.

In addition to creation and assignment functions, there also exist destruction and interrogation functions for most of the above constructs. Where applicable, these are defined by replacing `create` with `delete`, or by leaving off `set_`, respectively. It is never required to delete a data structure when it is no longer needed, but in extreme cases, when many calls are made to the creation functions, space may become tight.

A quick way to customize a decomposition is to create one using a predefined high-level routine, and then modify it. For example, one may want to use a three-dimensional diagonal multi-partition scheme, but partitions owned by processor 2 should be transferred to processor 7:

```
do    kp = 1, no_partitions(decomposition,3)
  do    jp = 1, no_partitions(decomposition,2)
    do    ip = 1, no_partitions(decomposition,1)
      p = partition_index(decomposition,ip,jp,kp)
      if (partition_owner(decomposition,p) .eq. 2) then
         call set_partition_owner(decomposition,7,p)
      end if
    end do
  end do
end do
```

Alternatively, when the particular location of the partition in the decomposition is irrelevant, the triple loop can be written as a single canonical loop over all partitions in the decomposition:

```
do    p = 1, total_no_partitions(decomposition)
  if (partition_owner(decomposition,p) .eq. 2) then
     call set_partition_owner(decomposition,7,p)
  end if
end do
```

Execution of `commit_decomposition` is necessary when construction of a decomposition is finalized. It serves to check consistency of the definition of the decomposition and to compute some information concerning communication schedules. It is possible to modify some of the data structures underlying a decomposition that make the actual decomposition invalid. For example, if the number of cuts in a certain dimension of the grid is changed, the partition ownership schedule contained in the decomposition can no longer be valid. In that case commit_decomposition will fail and a new decomposition must be constructed. If the decomposition is found to be valid but to not qualify as one of the special predefined types anymore, the special internal attribute is reset.

Once a decomposition has been formed, distributed variables can be associated with it. The function create_distribution creates a distributed array of some elementary data_type (MPI_integer, MPI_REAL, etc), whose storage is reserved at some specified start_address. The user also specifies the tensor rank of the variable, plus an array of numbers of components for each index of the rank. For example, setting rank equal to 2 and components equal to (3,3) defines a 3×3 matrix at each point of the grid. To accommodate stencil operations the user specifies a positive number of ghost_points. To determine whether enough space has been allocated for the local portion of the distributed variable, the user can request the number of units of the elementary data type needed, using storage_space.

Syntax of Fortran 77 distribution function:

```
      subroutine create_distribution(distribution,decomposition,data_type,
     $             start_address,rank,components,ghost_points)
      integer distribution, decomposition, data_type, rank,
              components(*), ghost_points
      <type>  start_address(*)

      integer function storage_space(decomposition)
      integer decomposition
```

Here <type> refers to a range of memory locations reserved for storage of elements of type data_type. By default, multiple partitions owned by the same processor are stored such that each partition takes up an equal amount of space. The layout is consistent with a storage declaration that allocates to each partition a subarray of identical dimensions. This will, in general, create gaps, which is wasteful. But it does allow uniform and simple declaration of complex distributed variables. Space can be conserved by calling the function compact, which eliminates any gaps, but necessitates the use of Charon access function offset (see Section 4) to determine where a particular partition starts in memory. Since every partition may have different subarray dimensions when compacted, suitable dimensioning statements may require calls to partition_size. Complete control over memory allocation is got by specifying explicitly where each partition p starts in memory, and what the subarray dimensions are. Such specifications, or calls to compact, must be made before any part of the distributed variable is used, because they affect important Charon functions, such as address, and value.

Syntax of Fortran 77 layout functions:

```
      subroutine compact(distribution)
      integer distribution
```

```
subroutine set_offset(distribution,location,p)
integer distribution, location, p

subroutine set_array_dimension(distribution,dimensions,p)
integer distribution, dimensions(*), p
```

Finally, we note that all grid, partition, decomposition and distribution creation and manipulation operations are global, which means that all processors in the corresponding communicator must call these routines with the same parameters.

4 Execution support tools

At the highest level of abstraction, Charon must present an interface that makes the transition from a serial to a *correct* parallel implementation simple and straightforward. The user need not be concerned about details of the domain decomposition, local and remote data, concurrency, communication, etc. Effectively, the top level programming tools support the Charon data distributions (as do the other levels), but hide the distribution aspects from the user. This is generally inefficient, but that is not a problem. In subsequent refinements performance improvements can be obtained, again making use of Charon tools.

We note that Charon does not provide an automatic code conversion capability. All parallelization is carried out by the user, who retains complete control over data lay-out and program flow. Hence, the necessary code changes must be kept at a minimum. For that purpose Charon offers execution support tools that simulate a single data space and a single thread of control. Assignments and control structures are exact images of the serial program, and the resulting code is executed by all processors; Charon simulates a single, replicated program counter. We use the following rationale for the implementation. Each element of a distributed variable has a unique *owner* processor, so it is most natural—and often least expensive—to employ an *owner-computes* rule: whenever an element of a distributed variable occurs on a left-hand side of an assignment, the processor who owns it is responsible for its evaluation. But since all processors execute the same code within the same control structure, we have to provide a mechanism to skip assignments to *nonlocal* memory locations; the replicated program counters 'pause' on all processors, except on the one executing the local assignment, and 'resume' collectively immediately thereafter. The obvious way to implement (nested) loops over distributed data structures is as follows:

1. Compute the intersection of the index sets of the loop and of the locally owned part of the distributed variable(s) on the left hand side of the assignment(s) in the loop body. This is the execution mask.

2. Execute the statements covered by the mask on each processor independently.

This would be the equivalent of a High Performance Fortran FORALL statement [2]. It is important to recognize that loops cannot be parallelized this way in general. Not only does it violate the principle of a simulated single program counter, it also constitutes a reordering or splitting of the original loop, which cannot be done with impunity in case of recurrences or other dependencies within the loop body or across iterations. Note also that locally owned index sets in Charon can be arbitrarily complex, due to the flexibility of the partitioning mechanism, and can generally not be described explicitly in the same loop structure as that of the serial program.

We use the following approach instead. Each assignment in the serial program is translated into a call to an **assign** routine, which takes as arguments a left hand side (an address) and a right hand side (a value). If the address is NULL (not reachable), no assignment takes place, and the statement is effectively not executed by the calling processor; it is masked. NULL addresses result from (unintentional) memory errors, and from assignments to nonlocal memory locations. The masking is obtained by using the function **address**, which returns an actual location for a local element of a distributed variable, and NULL for a nonlocal element. Masking alone is not sufficient, however, because the right hand side to be evaluated can also be arbitrarily complex, with possible contributions from local and remote memories. For this purpose the function **value** is introduced. It operates on distributed variables and always returns the proper value, or perhaps a 'conventional' segmentation fault in case of a programming error. No distinction is made between values returned by the function **value** and values of nondistributed variables and constants. All are *rvalues* in C terminology. Similarly, no distinction is made between addresses returned by the function **address**, and addresses of nondistributed variables. Both are *lvalues* in C terminology. In a correct program using only the high-level Charon tools, all rvalues always exist on each node, whereas lvalues are only defined if they are local. Alternatively, we may say that the highest level of abstraction of Charon only implements (implicitly invoked) remote gets, not puts. Implementation of remote gets does not require one-sided communication, since all processors call all **assign** routines and hence can cooperate in the assembly of each value to be assigned. Because rvalues must always be defined, **value** causes the processor that owns the particular data item to broadcast it to all other processors in the same communicator.

The names of each of the functions **assign**, **value**, and **address**, and many others, can be changed by the user by editing a dictionary before installing Charon. This allows for terser code, at the possible expense of reduced readability.

Syntax of (Fortran 77) global access functions (var constitutes a handle to a distributed variable):

```
subroutine assign(my_address,my_value)
<type> my_address, my_value

<type>  function value(var,index1,index2, ...)
integer var, index1, index2, ...

<type>  function address(var,index1,index2, ...)
integer var, index1, index2, ...
```

Observations:

- Fortran 77 does not make explicit distinction between rvalues and lvalues in declarations, so the definition of **assign** appears symmetric in its arguments. In Fortran 90, my_address would be declared with intent(out), and my_value with intent(in). Also in Fortran 90, my_address can be declared as a symbolic name, or as a pointer variable. In C or C++, my_address is a pointer and my_value is an actual value.

- The assign operator is overloaded; it can take values and addresses of any elementary type, as long as they are consistent (that is, value and address must refer to the same type). The user is responsible for the match. Errors cannot be detected by the compiler or runtime system.

- The generic function value specializes to real_value, integer_value, logical_value, double_precision_value and character_value, depending on the type of the distributed variable whose handle (var) it receives. Similar specializations hold for the generic function address. In addition, value and address allow variable-length parameter lists in order to accommodate distributed variables of differing dimensions. Integers index1, index2, ... are global indices with respect to the whole grid. Note that the result of the function address is used functionally as an lvalue by Charon, which is not possible in Fortran.

For the reasons listed above, assign, value and address are all implemented in C. However, they are callable from Fortran. Correctness (i.e. serial consistency) of a program utilizing only these routines is easily shown, even though Charon makes no assumptions about lock-step execution or other synchronization features of the runtime system, and does not pose any restrictions on data dependencies in the program. Each invocation of assign requires the cooperation of the processor that owns a referenced remote data element. Because all processors execute the same code, any update of such referenced remote data occurring logically before the value is requested must already have taken place before the request is registered and satisfied; synchronization is performed automatically. This is equivalent to realigning the replicated program counter. A side effect of the cooperative nature of implicitly invoked communications is that they must be issued as broadcast operations. A processor executing the value function must take active part in sending data, but cannot know which processor is the recipient until address has been evaluated. Both the address and the expression involving value are arguments of the assign routine, and Fortran semantics say that they may be executed in arbitrary order. Hence, the rvalue may need to be evaluated before the destination address is known, which implies that the rvalue be available to *all* processors in the communicator. A broadcast is required. If the lvalue is not a distributed variable—in other words, if it is a global variable—the address routine will not be used. Global variables are automatically self-consistent, because each processor assigns the same (broadcast) value to its local copy[1].

The simplest optimization that can be performed is to by-pass execution of assign statements involving remote data elements, so that no broadcasts need to take place. Charon is notified of this by the bracketing statements begin_local and end_local.

As an example Charon application we transform a serial code fragment that computes a distributed variable pr on a two-dimensional grid and counts the number of times it drops below zero.

```
count = 0
do    j = 1, nj
```

[1] Temporary variables inside loops are also global variables, and assignments to them will invoke broadcast operations if the right hand side expression contains distributed variables.

```
    do    i = 1, ni
      pr(i,j) = a(i,j)**2 - 1.0/b(i,j)
      if (pr(i,j) .lt. 0.0) count = count+1
    end do
  end do
```

Here is the first parallelized version. Handles to distributed variables are indicated by an underscore (_) suffix.

```
  count = 0
  do    j = 1, nj
    do    i = 1, ni
      call assign(address(pr_,i,j),
$                 value(a_,i,j)**2 - 1.0/value(b_,i,j))
      if (value(pr_,i,j) .lt. 0.0) count = count+1
    end do
  end do
```

We use the generic names for the Charon access functions for brevity. Notice that the structure of the parallel loop nest is identical to that of the original version, and that we have not made any assumptions about how the grid has been partitioned. The above loop nest is completely serialized, with only one processor making assignments to the distributed variable pr at any one time. Notice also that all processors execute the conditional statement for every iteration, causing a potentially large number of remote-memory accesses

In order to improve the performance of this loop while retaining the original structure, we make use of function point_owner to test whether a point in the decomposition (signified by the handle decomp) is assigned to the calling processor (identified by the variable my_rank. We only execute the loop body if the outcome is true. This means that only one processor will be incrementing the counter, but the final tally needs to be known to all processors. Since we now allow violation of the principle of a single program counter by prescribing different actions depending on the processor number, Charon can no longer automatically guarantee correctness of the program and some user intervention becomes necessary in the form of a reduction operation. The following version eliminates all extraneous synchronizations and redundant communications:

```
  counttmp = 0
  call begin_local(decomp)
  do    j = 1, nj
    do    i = 1, ni
      if (point_owner(decomp,i,j) .eq. my_rank) then
        call assign(address_(pr_,i,j),
$                   value(a_,i,j)**2 - 1.0/value(b_,i,j))
        if (value(pr_,i,j) .lt. 0.0) counttmp = counttmp+1
      end if
    end do
  end do
  call end_local(decomp)
  call MPI_allreduce(counttmp,count,1,MPI_INTEGER,MPI_SUM,
$                    MPI_COMM_WORLD,ierr)
```

It should be noted that this version of the loop nest is still much more expensive than a hand-coded message-passing version. This is due to the fact that function calls are made during every iteration. Moreover, all processors do need to perform the ownership test for all elements of the iteration index space. The next optimization step is obtained by restricting the iteration index space a priori. If we assume that each processor owns exactly one partition of the grid, the following code results:

```
  counttmp = 0
  call begin_local(decomp)
  do    j = own_low(decomp,2,1), own_high(decomp,2,1)
    do    i = own_low(decomp,1,1), own_high(decomp,1,1)
      call assign(address(pr_,i,j),
$                 value(a_,i,j)**2 - 1.0/value(b_,i,j))
      if (value(pr_,i,j) .lt. 0.0) counttmp = counttmp+1
    end do
  end do
  call end_local(decomp)
  call MPI_allreduce(counttmp,count,1,MPI_INTEGER,MPI_SUM,
$                    MPI_COMM_WORLD,ierr)
```

This code optimization exposes the domain decomposition. If the decomposition had consisted of an arbitrary number of grid partitions per processor, then the above loop nest would have changed as follows:

```
  counttmp = 0
  call begin_local(decomp)
  do    p = 1, own_no_partitions(decomp)
    do    j = own_low(decomp,2,p), own_high(decomp,2,p)
      do    i = own_low(decomp,1,p), own_high(decomp,1,p)
        call assign(address(pr_,i,j),
$                   value(a_,i,j)**2 - 1.0/value(b_,i,j))
        if (value(pr_,i,j) .lt. 0.0) counttmp = counttmp+1
      end do
    end do
  end do
  call end_local(decomp)
  call MPI_allreduce(counttmp,count,1,MPI_INTEGER,MPI_SUM,
$                    MPI_COMM_WORLD,ierr)
```

Notice that point indexing is still global, i.e. with respect to the global grid. The price for this convenience is the calls to **assign**, **address** and **value** in the loop body. We now eliminate these and write the complete final loop as:

```
  counttmp = 0
  do    p = 1, own_no_partitions(decomp)
    do    j = 1, own_partition_size(decomp,2,p)
      do    i = 1, own_partition_size(decomp,1,p)
        pr(i,j,p) = a(i,j,p)**2 - 1.0/b(i,j,p))
        if (pr(i,j,p) .lt. 0.0) counttmp = counttmp+1
      end do
    end do
  end do
  call MPI_allreduce(counttmp,count,1,MPI_INTEGER,MPI_SUM,
$                    MPI_COMM_WORLD,ierr)
```

Now there is no need anymore to place calls to the **begin/end_local** pair, because there are no more calls to the **value** routine. We have finally obtained a code fragment that is as efficient as a hand-coded message-passing version. It is also almost as complicated as a message-passing code, so it would appear that nothing has been gained. However, the important difference with other systems is that this optimized code may be freely combined with high-level unoptimized code fragments—even within the same subroutine—that do not contain any references to the domain decomposition. It is the programmer's responsibility to provide the proper declarations and dimension statements for the distributed variables. In the above example arrays pr, a, and b were declared using three indices. But it would have been equally legal to write the loop over the partitions owned by the calling processor as follows (assuming **compact** storage of the distributed arrays):

```
  counttmp = 0
  do    p = 1, own_no_partitions(decomp)
    si_pr = offset(pr_,p)
    si_a  = offset(a_ ,p)
    si_b  = offset(b_ ,p)
    ni_p  = own_partition_size(decomp,1,p)
    nj_p  = own_partition_size(decomp,2,p)
    call sub_loop(pr(si_pr),a(si_a),b(si_b),ni_p,nj_p,counttmp)
  end do
  call MPI_allreduce(counttmp,count,1,MPI_INTEGER,MPI_SUM,
$                    MPI_COMM_WORLD,ierr)
  ...

  subroutine sub_loop(pr,a,b,ni,nj,counttmp)
  integer ni, nj, i, j, counttmp
  real pr(0:ni+1,0:nj+1), a(ni,nj), b(ni,nj)
  do    j = 1, nj
    do    i = 1, ni
      pr(i,j) = a(i,j)**2 - 1.0/b(i,j))
      if (pr(i,j) .lt. 0.0) counttmp = counttmp+1
    end do
  end do
```

```
      return
      end
```

Here we have assumed that arrays a and b are defined without any ghost points, while pr has a border of size 1. The Charon functions local_partition_size and start_index are used to interrogate the layouts of the decomposition and the distributed arrays, respectively. For complete portability we may use the interrogation function no_ghost_points(var_handle) to avoid implicit assumptions about ghost points.

The above code fragment loops over the locally owned partitions in the canonical fashion, i.e. in the order in which Charon numbers the partitions internally. If a particular order of visits by sub_loop were required, for example because synchronizations need to be performed after each layer of partitions in the j-direction has been completed, we can use additional interrogation functions. We also make use of the function own_partition_index, which returns for a particular partition its sequence number on the calling processor if it is owned by that processor, or -1 otherwise.

```
      counttmp = 0
      do    jp = 1, no_partitions(decomp,2)
        do    ip = 1, no_partitions(decomp,1)
          p = own_partition_index(decomp,ip,jp)
          if (p .ge. 0) then
            si_pr = offset(pr_,p)
            si_a  = offset(a_ ,p)
            si_b  = offset(b_ ,p)
            ni_p  = own_partition_size(decomp,1,p)
            nj_p  = own_partition_size(decomp,2,p)
            call sub_loop(pr(si_pr),a(si_a),b(si_b),ni_p,nj_p,counttmp)
          end if
        end do
        call MPI_barrier(MPI_COMM_WORLD,ierr)
      end do
      call MPI_allreduce(counttmp,count,1,MPI_INTEGER,MPI_SUM,
     $                   MPI_COMM_WORLD,ierr)
      ...
```

We now move to the more complex example of operations on arrays of matrix variables involving recurrences. The serial code represents the forward-elimination phase of a set of independent block-tridiagonal linear equations. Each of the diagonals low, main, and up consists of (4×4)-blocks. The right hand side vector r, which will be overwritten by the solution, consists of (4×1)-blocks. The recurrence is in the i-direction

```
      do    j = 1, nj
        do    i = 1, ni-1
          call invert_overwrite(up(1,1,i,j),main(1,1,i,j))
          call vinvert_overwrite(r(1,i,j),main(1,1,i,j))
          call multiply_add(main(1,1,i+1,j),low(1,1,i+1,j),up(1,1,i,j))
```

```
    call vmultiply_add(r(1,i+1,j),low(1,1,i+1,j),r(1,i,j))
  end do
end do
...

subroutine invert_overwrite(mat1,mat2)
real    mat1(4,4), mat2(4,4), temp(4,4), pivot
! code that overwrites mat1 by mat2^{-1}*mat1
pivot = 1.0/mat1(1,1)
...
return
end

subroutine vinvert_overwrite(vec,mat)
real    vec(4), mat(4,4), temp(4,4), pivot
! code that overwrites vec by mat2^{-1}*vec
pivot = 1.0/mat(1,1)
...
return
end

subroutine multiply_add(mat1,mat2,mat3)
real    mat1(4,4), mat2(4,4), mat3(4,4)
integer n, m, k
! code that overwrites mat1 by mat1-mat2*mat3
do    n = 1, 4
  do    m = 1, 4
    do    k = 1, 4
      mat1(n,m) = mat1(n,m)-mat2(n,k)*mat3(k,m)
    end do
  end do
end do
return
end

subroutine vmultiply_add(vec1,mat,vec2)
real    vec1(4), vec2(4), mat(4,4)
integer n, k
! code that overwrites vec1 by vec1-mat*vec2
do    n = 1, 4
  do    k = 1, 4
    vec1(n) = vec1(n)-mat(n,k)*vec2(k)
  end do
end do
return
```

```
      end
```

The difficulty with this code fragment is that the statements that do the actual work are in subroutines that have no knowledge of the overall partitioned grid. They operate on addresses and neighboring memory locations that are passed through parameter lists. Hence, the strategy of translating every assignment into a call to the Charon **assign** subroutine does not work. The solution is to demand that (v)invert_overwrite and (v)multiply_add execute atomically, which means that there must be a single, known address that acts as the start of a region of lvalues *on the same processor*. No other values may be modified within the same subroutine. There may be several contiguous regions of rvalues, whose sizes must also be known at run time. We now translate the above calling program; (v)invert_overwrite and (v)multiply_add remain unchanged.

```
      do    j = 1, nj
        do    i = 1, ni-1
          call invoke(invert_overwrite,address(up_,1,1,i,j),1,
     $                mvalue(16,main_,1,1,i,j))
          call invoke(vinvert_overwrite,address(r_,1,i,j),1,
     $                mvalue(16,main_,1,1,i,j))
          call invoke(multiply_add,address(main_,1,1,i+1,j),2,
     $                mvalue(16,low_,1,1,i+1,j),mvalue(16,up_,1,1,i,j))
          call invoke(vmultiply_add,address(r_,1,i+1,j),2,
     $                mvalue(16,low_,1,1,i+1,j),mvalue(4,r_,1,i,j))
        end do
      end do
```

Syntax of Fortran 77 bulk access functions:

```
      subroutine invoke(subf,my_address,no_inbufs,my_values1,my_values2,...)
      external subf
      integer no_inbufs
      <type> my_address(*), my_values1(*), my_values2(*), ...

      <type> function mvalue(n,var,index1,index2,index3,....)
      integer n, var, index1, index2, index3, ...
```

The general-purpose routine **invoke** examines the argument **my_address** and determines which processor is responsible for the execution of subroutine **subf**, based on the owner-computes rule. All other processors will skip the invocation of **subf**, but they will cooperate in providing rvalues, as needed, through communications. Again, atomicity is assumed, i.e. the processor that owns the first element of the distributed variable in the **mvalue** argument list is also responsible for supplying subsequent elements. If the first element is local, the action of **mvalue** is similar to that of **value**. If remote, a broadcast operation requesting n data elements is initiated. Upon completion, the function **mvalue** points to the start of a buffer region containing the requested values. The actual number of bytes transferred depends on the specific data type **<type>** of the distributed variable. The subroutine **subf** is defined by the user. It is invoked by Charon as follows:

```
call subf(my_address,my_values1,my_values2,...)
```

Fortran does not provide information about the number of actual parameters with which a subroutine is called, so the user must indicate to subroutine **invoke** the number of separate regions of input values through the parameter **no_inbufs**. If the user-defined subroutine also takes constants or non-distributed variables as arguments, a slight complication arises, because Fortran does not distinguish between scalar and array arguments of subroutines. It does not allow the type of overloading through argument checking that C++ does. In Fortran all arguments get translated into addresses, that are then passed to the subroutine. If the argument is an expression, it gets evaluated and the address of the result is passed to the subroutine (cf. **value**). But **invoke** expects for each argument not scalars, but pointers to arrays. To coerce **invoke**'s arguments into this behavior without using **mvalue**, which is reserved for distributed arrays, use the function **gvalue**. **gvalue** applies to both global scalars and global arrays.

Notice again that in the translated code fragment no influence of the domain decomposition is visible. The situation changes when we start to optimize the code. First assume that the grid is partitioned in a stripwise fashion, such that all points on a grid line of constant j are within the same partition. The block-tridiagonal systems can be solved independently by all processors, without any communication. Hence, the first optimization is again obtained by using the **begin/end_local** pair. Skipping a few steps, we easily arrive at the following efficient code:

```
do    j = 1, own_partition_size(decomp,2,1)
  do    i = 1, ni-1
    call invert_overwrite(up(1,1,i,j),main(1,1,i,j))
    call vinvert_overwrite(r(1,i,j),main(1,1,i,j))
    call multiply_add(main(1,1,i+1,j),low(1,1,i+1,j),up(1,1,i,j))
    call vmultiply_add(r(1,i+1,j),low(1,1,i+1,j),r(1,i,j))
  end do
end do
```

The problem becomes more interesting when the grid is partitioned differently, for example because there are other conflicting recurrences in the program. Assume again that each processor owns one partition, but this time the partition does not stretch the whole width of the grid. We first force the solution algorithm to proceed along stripwise sections of the grid, but retain the convenience of the implicitly invoked remote gets.

```
do    ip = 1, no_partitions(decomp,1)
  do    jp = 1, no_partitions(decomp,2)
    p = partition_index(decomp,ip,jp)
    do    j = low(decomp,2,p), high(decomp,2,p)
      do    i = low(decomp,1,p), min(ni-1,high(decomp,1,p))
        call invoke(invert_overwrite,address(up_,1,1,i,j),1,
$                     mvalue(16,main_,1,1,i,j))
        call invoke(vinvert_overwrite,address(r_,1,i,j),1,
$                     mvalue(16,main_,1,1,i,j))
```

```
     call invoke(multiply_add,address(main_,1,1,i+1,j),2,
$          mvalue(16,low_,1,1,i+1,j),mvalue(16,up_,1,1,i,j))
     call invoke(vmultiply_add,address(r_,1,i+1,j),2,
$          mvalue(16,low_,1,1,i+1,j),mvalue(4,r_,1,i,j))
    end do
   end do
  end do
 end do
```

Observe that the original loop has been reordered, but that the recurrence relation is respected. Next, the need for frequent communications must be eliminated. Since the recurrence relation has a depth of only one, a border of ghost points of size one suffices for the distributed arrays. Copying ghost point values from neighboring partitions is accomplished by using function copy_faces.

Before the loop is entered, all ghost point values are initialized. During the execution of the loop nest a complication arises, because the set of four bulk assignment statements in the loop body straddles the boundaries of partitions. Whenever the assignments 'spill over' into the next partition, it would be preferable to write ghost point values, rather than move partition ownership to another processor to do the spilled-over assignment. The mechanism for causing this to happen is the bracketing pair begin/end_ghost_access(p), which forces ghost point values of partition p to be written, instead of values of points properly contained in other partitions (provided the ghost points exist). In addition, ghost point values are read, instead of requested through a communication or a local memory copy, again provided the ghost points exist. It is as if temporarily the partition owned by the calling processor has been enlarged by the ghost points As before, exactly one processor is responsible for performing the computation of each distributed array element, regardless of whether the begin/end_ghost_access calls are placed. After all ghost points have been written for a whole strip of partitions, the values are transferred in bulk to the neighboring processors by calling the function copy_ghost_faces.

Syntax of Fortran 77 face copy functions:

```
subroutine copy_faces(var,periodicity,thickness,dimension,cut,subset)
integer var, periodicity, thickness, dimension, cut, subset(2,*)

subroutine copy_ghost_faces(var,periodicity,thickness,dimension,
                            cut,subset)
integer var, periodicity, thickness, dimension, cut, subset(2,*)

subroutine copy_faces_all(var,periodicity,thickness, stencil_shape)
integer var, periodicity, thickness, stencil_shape

subroutine copy_ghost_faces_all(var,periodicity,thickness,stencil_shape)
integer var, periodicity, thickness, stencil_shape
```

Copy_faces copies values from the outermost 'layers' of partitions to the corresponding ghost points of adjacent partitions. thickness refers to the number of layers to be copied (smaller than or equal to the total thickness of the borders). If **thickness** is ALL, all layers are copied. If **dimension** equals p, copying takes place to neighboring partitions in the p^{th} coordinate direction, where p can be positive or negative. If copying is required in a single dimension but in both directions, write p+ALL. The parameter cut specifies the sequence number of the cut in the coordinate direction defined by dimension across which the copy operation is to take place. Setting cut equal to ALL selects all cuts simultaneously. If the copy operation is PERIODIC, the sequence number of the cut may be zero, or one more than the actual number of cuts present (in Fortran, where the default array starting index is 1). Either case will be interpreted as a periodic copy. If the operation is NONPERIODIC, such cuts are ignored. Additionally, we may restrict the copying to a subset of a particular face, indicated by the two-dimensional array subset. It lists, for each coordinate direction other than that normal to the face to be copied, the start and end coordinates of the points of the subset with respect to the global grid. Alternatively, use ALL for the starting index if all points along the cut are to be copied.

Often it will be useful to copy face values at all cuts in all dimensions and directions, especially in the case of explicit methods, where all off-processor information can be fetched beforehand. For this purpose the variation copy_faces_all is provided. It takes as an argument the stencil shape, which can have the values BOX or STAR (see [1]). The STAR shape ignores diagonal values and only copies between strongly connected partitions. BOX, which also copies values to weakly connected partitions, will result in a staged copying of data to reduce latency. Figure 1 shows several different applications of the copy_faces[_all] routines.

Copy_ghost_faces accomplishes the same as copy_faces, but copies values from ghost points to points properly owned by neighboring partitions.

Notice that both types of copy functions are data-parallel. All processors participate, in principle, but only operate on the data that they own (if a processor is responsible for neither sending nor receiving, it can safely skip the copy call). In fact, the result of copy operations is independent of the number of processors involved, but depends only on the number and lay-out of the partitions. This is true for all Charon operations defined so far; distributed programs can be simulated, tested and debugged, to a large extent, while using only a single processor. One may even use Charon exclusively to obtain blocked uniprocessor code that optimizes data locality. Hence, the following program will run correctly on any number of processors.

```
      call copy_faces(r_,NONPERIODIC,1,-1,ALL,ALL)
      call copy_faces(low_,NONPERIODIC,1,-1,ALL,ALL)
      call copy_faces(main_,NONPERIODIC,1,-1,ALL,ALL)
      do    ip = 1, no_partitions(decomp,1)
        do    jp = 1, no_partitions(decomp,2)
          p = partition_index(decomp,ip,jp)
          call begin_ghost_access(decomp,p)
          do    j = low(decomp,2,p), high(decomp,2,p)
            do    i = low(decomp,1,p), min(ni-1,high(decomp,1,p))
              call invoke(invert_overwrite,address(up_,1,1,i,j),1,
     $                    mvalue(16,main_,1,1,i,j))
```

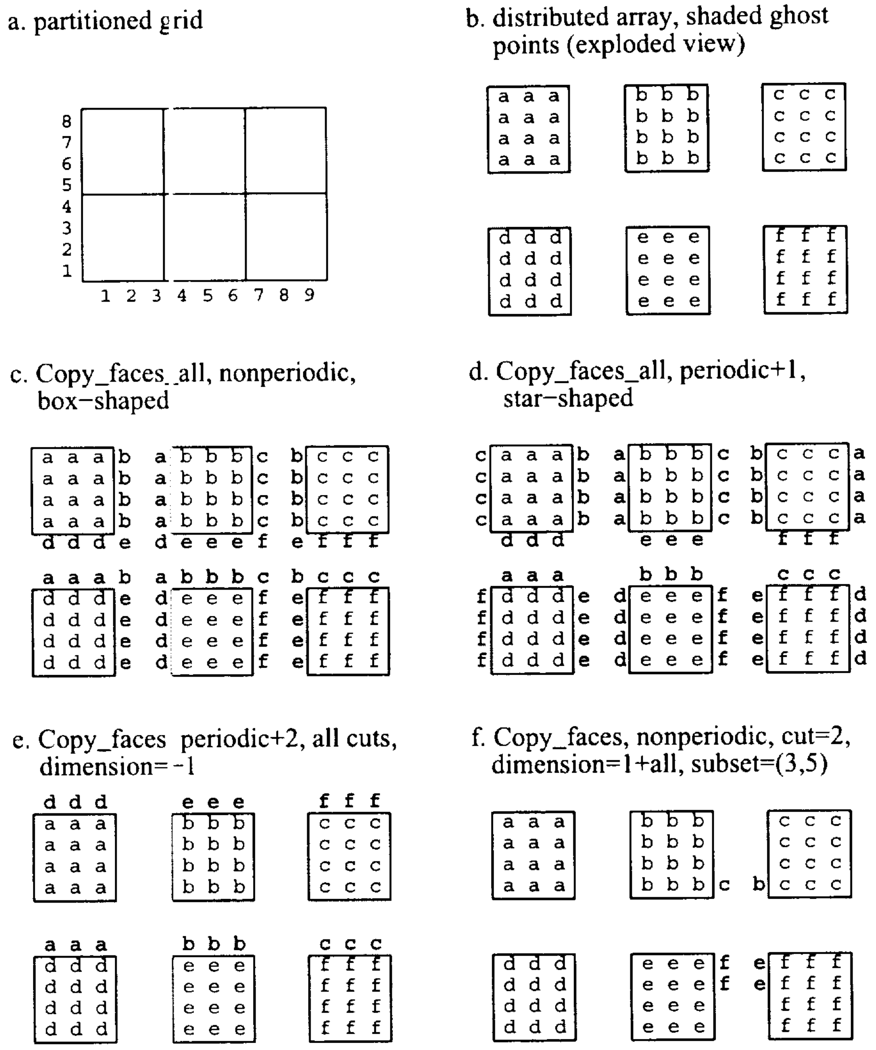
a. partitioned grid
8
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8 9
b. distributed array, shaded ghost points (exploded view)
c. Copy_faces_all, nonperiodic, box−shaped
d. Copy_faces_all, periodic+1, star−shaped
e. Copy_faces periodic+2, all cuts, dimension=−1
f. Copy_faces, nonperiodic, cut=2, dimension=1+all, subset=(3,5)

Figure 1: Applications of copy_faces and copy_faces_all to a distributed array.

```
          call invoke(vinvert_overwrite,address(r_,1,i,j),1,
     $                mvalue(16,main_,1,1,i,j))
          call invoke(multiply_add,address(main_,1,1,i+1,j),2,
     $                mvalue(16,low_,1,1,i+1,j),mvalue(16,up_,1,1,i,j))
          call invoke(vmultiply_add,address(r_,1,i+1,j),2,
     $                mvalue(16,low_,1,1,i+1,j),mvalue(4,r_,1,i,j))
        end do
      end do
      call end_ghost_access(decomp,p)
    end do
    call copy_ghost_faces(main_,NONPERIODIC,1,1,ip,ALL)
    call copy_ghost_faces(r_,NONPERIODIC,1,1,ip,ALL)
```

```
 end do
```

The above loop structure no longer requires the implicitly invoked communications. Consequently, we can relax the principle of the simulated single program counter and let each processor execute only its own part of the loop. But a poor load balance obtains, because only those processors who own partitions in the same strip of the grid can be active at the same time:

```
 call copy_faces(r_,NONPERIODIC,1,-1,ALL,ALL)
 call copy_faces(low_,NONPERIODIC,1,-1,ALL,ALL)
 call copy_faces(main_,NONPERIODIC,1,-1,ALL,ALL)
 call begin_local(decomp)
 do    ip = 1, no_partitions(decomp,1)
   do    jp = 1, no_partitions(decomp,2)
     p = partition_index(decomp,ip,jp)
     if (partition_owner(decomp,p) .eq. my_rank) then
       call begin_ghost_access(decomp,p)
       do    j = low(decomp,2,p), high(decomp,2,p)
         do    i = low(decomp,1,p), min(ni-1,high(decomp,1,p))
           call invoke(invert_overwrite,address(up_,1,1,i,j),1,
$                      mvalue(16,main_,1,1,i,j))
           call invoke(vinvert_overwrite,address(r_,1,i,j),1,
$                      mvalue(16,main_,1,1,i,j))
           call invoke(multiply_add,address(main_,1,1,i+1,j),2,
$                      mvalue(16,low_,1,1,i+1,j),mvalue(16,up_,1,1,i,j))
           call invoke(vmultiply_add,address(r_,1,i+1,j),2,
$                      mvalue(16,low_,1,1,i+1,j),mvalue(4,r_,1,i,j))
         end do
       end do
       call end_ghost_access(decomp,p)
     end if
   end do
   call copy_ghost_faces(main_,NONPERIODIC,1,1,ip,ALL)
   call copy_ghost_faces(r_,NONPERIODIC,1,1,ip,ALL)
 end do
 call end_local(decomp)
```

4.1 Multi-partition version of tri-diagonal solver

The load balance of the above loop nest can be improved by selecting another domain decomposition. If each processor receives not one but several partitions of the grid, arranged according to the diagonal multi-partition scheme [5, 6], the loop automatically becomes load balanced. Eliminating the invoke references, we obtain the final version of the code:

```
 call copy_faces(r_,NONPERIODIC,1,-1,ALL,ALL)
 call copy_faces(low_,NONPERIODIC,1,-1,ALL,ALL)
```

```
      call copy_faces(main_,NONPERIODIC,1,-1,ALL,ALL)
      do    ip = 1, no_partitions(decomp,1)
        do    jp = 1, no_partitions(decomp,2)
          p = own_partition_index(decomp,ip,jp)
          if (p .gt. 0) then
            do    j = 1, own_partition_size(decomp,2,p)
              ihigh = own_partition_size(decomp,1,p)
              if (ip .eq. no_partitions(decomp,1)) ihigh = ihigh-1
              do    i = 1, ihigh
                call invert_overwrite(up(1,1,i,j,p),main(1,1,i,j,p))
                call vinvert_overwrite(r(1,i,j,p),main(1,1,i,j,p))
                call multiply_add(main(1,1,i+1,j,p),low(1,1,i+1,j,p),
     $                            up(1,1,i,j,p))
                call vmultiply_add(r(1,i+1,j,p),low(1,1,i+1,j,p),r(1,i,j,p))
              end do
            end do
          end if
        end do
        call copy_ghost_faces(main_,NONPERIODIC,1,1,ip,ALL)
        call copy_ghost_faces(r_,NONPERIODIC,1,1,ip,ALL)
      end do
```

4.2 Pipelined uni-partition version of tri-diagonal solver

If multi-partitioning is not feasible, performance of the loop nest can still be improved by pipelining the uni-partition solver. We will assume for simplicity that the number of grid points in the j-direction is divided evenly among the processors, and that the size of each partition in the j-direction is divisible by the pipeline grouping factor npipe. Otherwise, some preconditioning would be necessary.

```
      call copy_faces(r_,NONPERIODIC,1,-1,ALL,ALL)
      call copy_faces(low_,NONPERIODIC,1,-1,ALL,ALL)
      call copy_faces(main_,NONPERIODIC,1,-1,ALL,ALL)
      call begin_local(decomp)
      do    ip = 1, no_partitions(decomp,1)
        do    jp = 1, no_partitions(decomp,2)
          p = partition_index(decomp,ip,jp)
          no_stages = (high(decomp,2,p)-low(decomp,2,p)+1)/npipe
          do    stage = 1, no_stages
            subset(1,1) = low(decomp,2,p)+(stage-1)*npipe
            subset(2,1) = subset(1,1)-1+npipe
            if (partition_owner(decomp,p) .eq. my_rank) then
              call begin_ghost_access(decomp,p)
              do    j = subset(1,1), subset(2,1)
                do    i = low(decomp,1,p), min(ni-1,high(decomp,1,p))
                  call invoke(invert_overwrite,address(up_,1,1,i,j),1,
$                       mvalue(16,main_,1,1,i,j))
            call invoke(vinvert_overwrite,address(r_,1,i,j),1,
$                       mvalue(16,main_,1,1,i,j))
            call invoke(multiply_add,address(main_,1,1,i+1,j),2,
$                       mvalue(16,low_,1,1,i+1,j),mvalue(16,up_,1,1,i,j))
            call invoke(vmultiply_add,address(r_,1,i+1,j),2,
$                       mvalue(16,low_,1,1,i+1,j),mvalue(4,r_,1,i,j))
          end do
        end do
        call end_ghost_access(decomp,p)
      end if
      call copy_ghost_faces(main_,NONPERIODIC,1,1,ip,subset)
      call copy_ghost_faces(r_,NONPERIODIC,1,1,ip,subset)
    end do
  end do
end do
call end_local(decomp)
```

The above loop deviates from what would usually be obtained by programming a pipelined equation solver from scratch. Most significantly, there is only one copy of the main loop body; no special cases have to be distinguished for processors at the begin or end of the pipeline. Moreover, synchronization appears in a natural place, namely after the program text that finishes a stage of the pipeline. This is a fringe benefit of the coding style encouraged by Charon. The user is led to approach the numerical problem at hand from the perspective of the overall data space and instruction stream, not just from the subset owned by each individual processor. Equally important is the Charon concept of data-parallel communications. All processors call each communication routine, in principle, but action and/or synchronization is required only by those processors that own or need the data that is communicated.

Finally, we revert to accessing the distributed-array elements directly. This eliminates the overhead of the many function calls to invoke, address, and mvalue. In addition, we rearrange the order in which the partitions are visited. Pipelining inhibits copying interface data along entire cuts, and copying only a few numbers at a time can lead to a significant overhead if all processors must execute each instance of copy_ghost_faces. So instead, we write the loop nest as a set of independent pipelines, one for each strip of partitions in the i-direction. To accomplish this we make use of the function own_partition_coordinate, which returns for the p^{th} partition owned by the calling processor the partition index in a specified coordinate direction (i.e. the sequence number of the strip of partitions). The copy operation after completion of each pipeline segment will finish correctly and without deadlock, because all processors that are involved in the communication are in the same strip, and hence call the copy routine.

```
call copy_faces(r_,NONPERIODIC,1,-1,ALL,ALL)
call copy_faces(low_,NONPERIODIC,1,-1,ALL,ALL)
call copy_faces(main_,NONPERIODIC,1,-1,ALL,ALL)
do    jp = 1, no_partitions(decomp,2)
```

```
  if (own_partition_coordinate(decomp,1,2) .eq. jp) then
    do    ip = 1, no_partitions(decomp,1)
      p = partition_index(decomp,ip,jp)
      no_stages = (high(decomp,2,p)-low(decomp,2,p)+1)/npipe
      do    stage = 1, no_stages
        subset(1,1) = low(decomp,2,p)+(stage-1)*npipe
        subset(2,1) = subset(1,1)-1+npipe
        if (partition_owner(decomp,p) .eq. my_rank) then
          do    j = 1+(stage-1)*npipe, stage*npipe
            ihigh = partition_size(decomp,1,p)
            if (ip .eq. no_partitions(decomp,1)) ihigh = ihigh-1
            do    i = 1, ihigh
              call invert_overwrite(up(1,1,i,j),main(1,1,i,j))
              call vinvert_overwrite(r(1,i,j),main(1,1,i,j))
              call multiply_add(main(1,1,i+1,j),low(1,1,i+1,j),up(1,1,i,j))
              call vmultiply_add(r(1,i+1,j),low(1,1,i+1,j),r(1,i,j))
            end do
          end do
        end if
        call copy_ghost_faces(main_,NONPERIODIC,1,1,ip,subset)
        call copy_ghost_faces(r_,NONPERIODIC,1,1,ip,subset)
      end do
    end do
  end if
end do
```

4.3 Additional solver optimizations

Although the final multi-partition and pipelined uni-partition codes are quite efficient, there are some other optimizations that users might consider. First, there is no strict necessity to use ghost points for this application. They are used to satisfy the remote data requirements of the solver implicitly by duplicating such data in the location where they are expected by the calling processor. But explicit message passing could also have been used (the lowest level of abstraction in Charon), in which send and receive buffers are managed by the user. Buffer data can then be integrated in the computation directly as it arrives, without first being stored in the ghost point locations. This is not necessarily faster than using ghost points, but it saves space. Second, a certain overhead is incurred by letting processors perform a loop over partitions they do not own, instead of focusing on those they do own. Since no computational work is done on nonlocal partitions, this overhead is small, except in the case of extremely fine partitioning. Then the multi-partition method can be further improved by computing in advance which partition in each strip of partitions is owned by the calling processor Third, if explicit message passing were used, some message aggregation would be possible by fusing co-located calls to the copy routines. Because Charon provides the mechanism to construct coarse-grained parallel programs with relative ease, latency reduction through such fusings is generally minimal. These optimizations should only be considered if absolutely required, since they increase the programming complexity significantly.

A potentially more useful optimization is obtained by using asynchronous communication calls icopy_faces, icopy_faces_all, etc. Such calls return immediately without blocking on the sending or receiving side. A copy_wait call must be issued at the point in the program where the expected data is actually used.

4.4 Data redistribution

Certain applications feature a succession of different and very strong data dependencies during the course of the computation. Such programs may benefit from a dynamic redistribution of part of their data to reduce the frequency of remote-memory accesses. This is accomplished by the routine redistribute. In most cases redistribution results in a global exchange of data, which is an expensive operation that should be applied with care. If possible, the asynchronous version iredistribute should be used. The routine assumes that both the source and the destination distributed variables have been predefined. Obviously, redistribution can only take place between distributed variables of the same tensor rank and vector dimensions, defined on the same grid. Increased efficiency is obtained for variables that are also based on the same partition, since this reduces the fragmentation of the packets of data to be communicated between processors. If space for the two distributions (partially) coincides, Charon treats the redistribution as an *in situ* operation, which is usually more expensive than in the case of spatially disjoint copies.

Syntax of Fortran 77 redistribution function:

```
subroutine redistribute(to_var,from_var)
integer to_var, from_var
```

5 Charon utilities

The previous sections described the basic Charon functions that can be used to write parallel programs in a piecemeal fashion. However, for the conversion of legacy codes some more utilities are needed. Common practice in the writing of scientific computing codes includes overindexing, and the introduction of lower-dimensional array segments. Overindexing is especially useful on vector computers, where it serves to increase the vector length of inner loops. Lower-dimensional array segments are used as scratch space, or to provide a convenient local reference to a higher-dimensional array. We will show how to implement these three techniques using Charon.

5.1 Specialized and generalized distributions

Lower-dimensional segments of distributed arrays are called *slices*. They are defined using the command create_slice. Since they use the same space as the arrays from which they are 'carved', no pointers to memory locations are necessary. The syntax of the command is as follows:

```
subroutine create_slice(var_slice,var,no_dims,indices)
integer var_slice, var, no_dims, indices(*)
```

The parameter no_dims indicates that dimensions 1 through no_dims of the original distributed array (signified by var) are retained. The array indices contains an ordered list of dropped—and hence constant—indices of the parent array. If the parameter var_slice refers to an existing, valid slice, that old slice definition is deleted and overwritten by the new one. This is consistent with the programming practice of sweeping over higher-dimensional arrays in a slicewise fashion. Depending on the size and the number of operations on the slice, calls to create_slice may represent a significant overhead. Higher efficiency can be achieved by creating and storing the slices only once in a preprocessing step. All operations on partitions (other than those that change the layout) can also be applied to slices. A slice will generally consist of a number of segments of partitions contained in the parent distributed array. These segments inherit their local sequence number from the corresponding partition.

Arrays of lower dimension than that of the grid that are used as scratch space are defined using a generalization of the create_distribution routine. This results, as in the case of create_slice, in the creation of a distribution.

```
      subroutine create_generalized_distribution(var,decomposition,data_type,
     $           start_address,rank,components,ghost_points,index,
     $           leading_position)
      integer var, decomposition, data_type, rank, components(*), ghost_points,
     $        index(*), leading_position
      <type>  start_address(*)
```

The value of index(i) indicates which index of the subarray corresponds to grid dimension i. If the value is negative, say -p, then that grid dimension is excluded from the generalized distribution, and its index is fixed at p. create_generalized_distribution can also be used to create plain distributions by setting index(i) equal to i for all dimensions of the grid. Permutations of grid indices are obtained by making index a proper permutation of the grid dimensions. Finally, the user can choose the relative positions of grid and tensor indices. By setting the parameter leading_position to TENSOR—the default for the standard create_distribution routine—the tensor indices are the fastest varying components of the distributed variable (in Fortran). Choosing GRID makes the grid indices vary fastest.

The technique of overindexing is the most complicated to accommodate, because it relies heavily on the implicitly defined storage format of Fortran arrays. It can also be relatively expensive when applied carelessly to loops that run over multiple partitions, and should generally be avoided in parallel programs. But since it is widely practiced in traditional scientific computing, some support for it must be provided. The Charon answer is to *fuse* a number (no_dims) of array indices of a distribution, thus creating an alias for the original distribution. The result again obeys all the rules for distributions.

```
      subroutine create_fused_distribution(var_alias,var,no_dims)
      integer var_alias, var, no_dims
```

As an example, if two dimensions (the first two) of a four-dimensional distributed array are fused, the variable is henceforth indexed using only three indices. Note that contiguous array elements in the serial code may not be contiguous anymore in the distributed array, even when they are within the same partition, due to the existence of ghost points. The Charon functions `value` and `address` will take proper care of this, and the results will be as expected, as if there were no ghost points. But the step from the high-level Charon code to the lower-level concurrent code with direct array access may no longer be as straightforward as before.

5.2 Irregular remote data requests

So far we have only discussed the facilitation of regular remote array accesses through the Charon `copy_faces` routines. But sometimes it is necessary to make reference to remote data in an arbitrary pattern. For this purpose Charon provides a device called `copy_tile`, which places in the memory of particular processors a copy of a Cartesian subset of a distributed variable.

```
subroutine copy_tile(var,comm,start_address,subset)
integer var, comm, subset(2,*)
<type>  start_address(*)
```

As all Charon communication routines, `copy_tile` is called by all processors in the communicator for which the distributed variable is defined. If the calling processor is a member of communicator `comm`, then a copy of the Cartesian subset defined by $\prod_{i=1}^{n}[subset(1,i), subset(2,i)]$ is placed in a buffer at `start_address`. An asynchronous, nonblocking version is also available under the name of `icopy_tile`.

There are certain similarities between `copy_tile` and the remote data requests in the Global Arrays package [3], as both mechanisms allow subsets of global data to be gathered. However, in the case of Global Arrays, such subsets are restricted to two-dimensional subsets of matrices, whereas in Charon subsets of arbitrary dimension are allowed. Another difference is that in Charon copying is a collective operation, and no explicit synchronization is required, except in the case of asynchronous transfers. By contrast, Global Arrays uses one-sided communication, which always requires synchronization.

5.3 Parallel I/O

Using the newly defined parallel I/O subset of MPI 2 [7], distributed variables can be written to a file in a single step, as was demonstrated in the multi-partition parallel flow solver RANS-MP [6]. Syntax: To be determined.

6 Conclusions

A practical design for a system to accommodate piecemeal conversion of serial legacy codes or designs to efficient distributed-memory message-passing programs has been presented. Although Charon targets complicated structured-grid applications, its principle extends to other areas as well. What is required is a set of functions that gives the user global access to elements of distributed data structures, in the vein of the Charon structured-grid functions `value` and `address`, supplemented with a set of functions to create and manipulate such data structures. The latter, as well as suitable data-parallel communication functions, are likely to be specific to the application area.

Whereas Charon does not itself comprise an automatic parallelization facility, it is amenable to the use of such tools. In principle, all the user need do is determine the distributions of large arrays. Using these as inputs, it can be inferred automatically where and how to place calls to `assign`, `invoke`, `address`, `value`, and `mvalue`. Subsequent tuning is then carried out by the programmer.

Finally, we summarize the features that distinguish Charon from systems that are based exclusively on (semi-)automatic program translation. Charon:

- gives the user complete control over data distribution, including advanced schedules such as multi-partitioning.
- gives the user complete control over memory usage and data layout within each processor, including coincident arrays and hand-tuned padding.
- gives the user complete control over granularity of the program through the flexible communication specification mechanisms of `copy_faces/tile`.
- allows redistribution of distributed arrays.
- allows easy incremental program change; no new analysis is needed when modules or statements are added, and previous tunings are never lost.
- poses no restrictions on programming model; although the Charon tools facilitate SPMD-style programming, use of different communicators provides the flexibility to create different contexts, and message-passing calls can always be used to support true MIMD programs.

The disadvantage of this flexibility is that hand-coding is involved, and that some knowledge of the program structure and data dependencies is required.

References

[1] S. Balay, W.D. Gropp, L. Curfman McInnes, B.F. Smith, *Efficient management of parallelism in object-oriented numerical software libraries*, Modern Software Tools in Scientific Computing, E. Arge, A.M. Bruaset, H.P. Langtangen, Ed., Birkhauser Press, 1997

[2] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, M. Zosel, *The High Performance Fortran Handbook*, MIT Press, Cambridge, MA, 1994

[3] J. Nieplocha, R.J. Harrison, R.J. Littlefield, *The global array programming model for high performance scientific computing*, SIAM News, Vol. 28, No. 7, August-September 1995

[4] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, *MPI: The Complete Reference*, MIT Press, 1995.

[5] R.F. Van der Wijngaart, *Charon toolkit for parallel, implicit structured-grid computations: Literature survey and conceptual design*, NAS Report xxx, NASA Ames Research Center, Moffett Field, CA, 1997

[6] R.F. Van der Wijngaart, M. Yarrow, M.H. Smith, *An architecture-independent parallel implicit flow solver with efficient I/O*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997

[7] MPI Forum, *MPI-2: Extensions to the Message-Passing Interface*,
URL: "http://www.mcs.anl.gov/Projects/mpi/mpi2/mpi2-report/mpi2-report.html"